Chapter TWENTY-FOUR
NIO.2

Exam Objectives

Use Path interface to operate on file and directory paths.
Use Files class to check, read, delete, copy, move, manage metadata of a file or directory.

NIO.2

In the last chapter, we reviewed the classes of the `java.io` package.

In the first versions of Java, this package, especially the `File` class, provided support for file operations. However, it had some problems, like lacking functionality and limited file attribute support.

For that reason, Java 1.4 introduced the NIO (Non-blocking Input/Output) API in the package `java.nio` implementing new functionality like channels, buffering, and new charsets.

However, this API didn't entirely solve the problems with the `java.io` package, so in Java 7, the NIO.2 API was added in the `java.nio.file` package (actually, since this is a new package, NIO.2 is not an update to the NIO API, besides, they focus on different things).

NIO.2 provides better support for accessing files and the file system, symbolic links, interoperability, and exceptions among others.

The primary classes of `java.nio.file`, `Path`, `Paths`, and `Files`, are intended to provide an easier way to work with files and to be a replacement for the `java.io.File` class.

These classes will be the focus of this chapter. Let's start with `Path` and `Paths`.

The Path interface

In the previous chapter, we also reviewed concepts like file systems and paths.

Well, the `Path` interface defines an object that represents the path to a file or a directory.

When you think about the fact that a path varies between different file systems, it makes sense that `Path` is an interface. Thanks to this, Java transparently handles different implementations between platforms.

For example, here are some differences between Windows-based and Unix-based systems:

- Windows-based systems are not case sensitive while Unix-based systems are.
- In Windows-based systems, paths are separated by backslashes. In Unix-based systems, by forward slashes.
- In Windows-based systems, the root path is a drive letter (generally `c:\`). In Unix-based systems, it's a forward slash (`/`).
- Because of that, in Windows-based systems, an absolute path starts with a drive letter (like `c:\temp\file.txt`). In Unix-based systems, it starts with a forward slash (like `/temp/file.txt`).

Since `Path` is an interface and Java handles its implementations, we have to use a utility class to create `Path` instances.

`java.nio.file.Paths` is this class. It provides two methods to create a `Path` object:

```
static Path get(String first, String... more)
static Path get(URI uri)
```

Be very careful with the names:

Path is the **interface** with methods to work with paths.

Paths is the **class** with `static` methods to create a `Path` object.

With the first version of `Paths.get()` you can create a `Path` object in these ways:

```
// With an absolute path in windows
Path pathWin = Paths.get("c:\\temp\\file.txt");
// With an absolute path in unix
Path pathUnix = Paths.get("/temp/file.txt");
// With a relative path
Path pathRelative = Paths.get("file.txt");
//Using the varargs parameter
// (the separator is inserted automatically)
Path pathByParts = Paths.get("c:", "temp", "file.txt");
```

With the second version, you have to use a `java.net.URI` instance. Since we're working with files, the URI schema must be `file://` :

```
try {
    Path fileURI = Paths.get(new URI("file:///c:/temp/file.txt"));
} catch (URISyntaxException e) {
    //This checked exception is thrown by the URI constructor
}
```

If you don't want to catch `URISyntaxException` , you can use the `static` method `URI.create(String)` . It wraps the `URISyntaxException` exception in an `IllegalArgumentException` (a subclass of `RuntimeException`):

```
Path fileURI = Paths.get(URI.create("file:///c:/temp/file.txt"));
```

Notice the three slashes. `file:///` represents an absolute path (the `file://` schema plus another slash for the root directory). We can test this with the help of the `toAbsolutePath()` method, which returns the absolute path representation of a `Path` object:

```
Path fileURI = Paths.get(URI.create("file:///file.txt"));
System.out.println(fileURI.toAbsolutePath());
```

This will print either:

```
C:\file.txt // in Windows-based systems
/file.txt // Or in Unix-based systems
```

We can also create a `Path` from a `File` and vice-versa:

```
File file = new File("/file.txt");
Path path = file.toPath();

path = Paths.get("/file.txt");
file = path.toFile();
```

And just to make clear that the `Path` instance is system-dependent, let me tell you that `Paths.get()` is actually equivalent to:

```
Path path = FileSystems.getDefault().getPath("c://temp");
```

As you can see from the examples, the absolute path representation of a `Path` object has a root component (either `c:\` or `/`) and a sequence of names separated by a (forward or backward) slash.

These names represent the directories needed to navigate to the target file or directory. The last name in the sequence represents the name of the target file or directory.

For example, the elements of the path `c:\temp\dir1\file.txt` (or its Unix equivalent, `/temp/dir1/file.txt`) are:

```
Root: c:\ (or /)
Name 1: temp
Name 2: dir1
Name 3: file.txt
```

The `Path` object has some methods to get this information. Except for `toString()` and `getNameCount()` , each of these methods returns a `Path` object):

```
Path path = Paths.get("C:\\temp\\dir1\\file.txt");
// Or Path path = Paths.get("/temp/dir1/file.txt");
System.out.println("toString(): " + path.toString());
System.out.println("getFileName(): " + path.getFileName());
System.out.println("getNameCount(): " +path.getNameCount());
// Indexes start from zero
System.out.println("getName(0): " + path.getName(0));
System.out.println("getName(1): " + path.getName(1));
System.out.println("getName(2): " + path.getName(2));
// subpath(beginIndex, endIndex) from beginIndex to endIndex-1
System.out.println("subpath(0,2): " + path.subpath(0,2));
System.out.println("getParent(): " + path.getParent());
System.out.println("getRoot(): " + path.getRoot());
```

The output:

```
toString(): C:\temp\dir1\file.txt // Or /temp/dir1/file.txt
getFileName(): file.txt
getNameCount(): 3
getName(0): temp
getName(1): dir1
getName(2): file.txt
subpath(0,2): temp\dir1 // Or temp/dir1
getParent(): C:\temp\dir1 // Or /temp/dir1
getRoot(): C:\ // Or /
```

Passing an invalid index to `getName()` and `subpath()` will throw an `IllegalArgumentException` (a `RuntimeException`).

If the path is specified as a relative one (and assuming this code is executed from the `c:\temp` directory):

```
Path path = Paths.get("dir1\\file.txt");// Or dir1/file.txt
System.out.println("toString(): " + path.toString());
System.out.println("getFileName(): " + path.getFileName());
System.out.println("getNameCount(): " + path.getNameCount());
System.out.println("getName(0): " + path.getName(0));
System.out.println("getName(1): " + path.getName(1));
System.out.println("subpath(0,2): " + path.subpath(0,2));
System.out.println("getParent(): " + path.getParent());
System.out.println("getRoot(): " + path.getRoot());
```

The output:

```
toString(): dir1\file.txt // Or dir1/file.txt
getFileName(): file.txt
getNameCount(): 2
getName(0): dir1
getName(1): file.txt
subpath(0,2): dir1\file.txt // Or dir1/file.txt
getParent(): dir1
getRoot(): null
```

When working with paths, you can use:

- `.` to refer to the current directory
- `..` to refer to the parent directory

For example:

```
// refers to /temp/file.txt
Path p1 = Paths.get("/temp/./file.txt");
// refers to /temp//file.txt
Path p2 = Paths.get( "/temp/dir1/../file.txt");
```

In these cases, you can use the `normalize()` method to remove redundancies like `.` and `..` (in other words, to "normalize" it):

```
Path path = Paths.get("/temp/dir1/../file.txt");
System.out.println(path); // /temp/dir1/../file.txt
Path path2 = path.normalize();
System.out.println(path2); // /temp/file.txt
```

This method does not access the file system to know if a file exists, so removing `..` and a preceding name from a path may result in a path that no longer references the original file. This can happen when that previous name is a symbolic link (a reference to another file).

It's better to use the `toRealPath()` method:

```
Path toRealPath(LinkOption... options) throws IOException
```

This method does the following:

- If `LinkOption.NOFOLLOW_LINKS` is passed as an argument, symbolic links are not followed (by default it does).
- If the path is relative, it returns an absolute path.
- It returns a `Path` with redundant elements removed (if any).

We can combine two paths. There are two cases.

First case. If we have an absolute path and we want to combine it with a second path that doesn't have a root element (a partial path), the second path is appended:

```
Path path = Paths.get("/temp");
System.out.println(path.resolve("newDir")); // /temp/newDir
```

Second case. If we have a partial or relative path, and we want to combine it with an absolute path, this absolute path is returned:

```
Path path = Paths.get("newDir");
System.out.println(path.resolve("/temp")); // /temp
```

`relativize()` is another interesting method.

`path1.relativize(path2)` is like saying *give me a path that shows how to get from path1 to path2*.

For example, if we are in directory `/temp` and we want to go to `/temp/dir1/subdir` , we have to go first to dir1 and then to subdir:

```
Path path1 = Paths.get("temp");
Path path2 = Paths.get("temp/dir1/file.txt");
Path path1ToPath2 = path1.relativize(path2); // dir1/file.txt
```

If the paths represent two relatives paths without any other information, they are considered siblings, so you have to go to the parent directory and then go to the other directory:

```
Path path1 = Paths.get("dir1");
Path path1ToPath2 = path1.relativize(Paths.get("dir2")); // ../dir2
```

Notice that both examples use relative paths.

If one of the paths is an absolute path, a relative path cannot be constructed because of the lack of information and a `llegalArgumentException` will be thrown.

If both paths are absolute, the result is system-dependent.

`Path` implements the `Iterable` interface so you can do something like this:

```
Path path = Paths.get("c:\\temp\\dir1\file.txt");
for(Path name : path) {
    System.out.println(name);
}
```

The output:

```
temp
dir1
file.txt
```

`Path` implements the `Comparable` interface and the `equals()` method to test two paths for equality.

`compareTo()` compares two paths lexicographically. It returns:

- Zero if the argument is equal to the path,
- A value less than zero if this path is lexicographically less than the argument, or
- A value greater than zero if this path is lexicographically greater than the argument.

The `equals()` implementation is system-dependent (for example, it's case insensitive on Windows systems). However, it returns `false` if the argument is not a `Path` or if it belongs to a different file system.

In addition, the methods `startsWith()` and `endsWith()` both test whether a path begins or ends with some `String` (in this case, the methods return `true` only if the string represents an actual element) or `Path` . So given:

```
Path absPath = Paths.get("c:\\temp\\dir1\\file.txt");
Path relPath = Paths.get("temp\\dir1\\file.txt");
```

```
boolean startsWith(Path other)
absPath.startsWith(Paths.get("c:\\temp\\file.txt")); // false
absPath.startsWith(Paths.get("c:\\temp\\dir1\\img.jpg")); // false
absPath.startsWith(Paths.get("c:\\temp\\dir1\\")) // true
absPath.startsWith(relPath); // false

boolean startsWith(String other)
relPath.startsWith("t"); // false
relPath.startsWith("temp"); // true
relPath.startsWith("temp\\d"); // false
relPath.startsWith("temp\\dir1"); // true

boolean endsWith(Path other)
absPath.endsWith("file.txt"); // true
absPath.endsWith("d:\\temp\\dir1\\file.txt"); // false
relPath.endsWith(absPath); // false

boolean endsWith(String other)
relPath.endsWith("txt"); // false
relPath.endsWith("file.txt"); // true
relPath.endsWith("\\dir1\\file.txt"); // false
relPath.endsWith("dir1\\file.txt"); // true
```

These methods don't take into account trailing separators, so if we have the `Path` `temp/dir1` , invoking, for example, `endsWith()` with `dir1/` , it returns `true` .

The Files class

The `java.nio.file.Files` class has `static` methods for common operations on files and directories. In contrast with the `java.io.File` class, all methods of `Files` work with `Path` objects (so don't confuse `File` and `Files`).

For example, we can check if a path actually exists (or doesn't exist) with the methods:

```
static boolean exists(Path path, LinkOption... options)
static boolean notExists(Path path, LinkOption... options)
```

If `LinkOption.NOFOLLOW_LINKS` is present, symbolic links are not followed (by default they are).

We can check if a path is readable (it's not if the file doesn't exist or if the JVM doesn't have the privileges to access it):

```
static boolean isReadable(Path path)
```

We can check if a path is writable (it's not if the file doesn't exist or if the JVM doesn't have the privileges to access it):

```
static boolean isWritable(Path path)
```

We can check if a file exists and is executable:

```
static boolean isExecutable(Path path)
```

Or even check if two paths refer to the same file (useful if one path represents a symbolic link). If both `Path` objects are equal then this method returns `true` without checking if the file exists:

```
static boolean isSameFile(Path path,
                          Path path2) throws IOException
```

To read a file, we can load the entire file into memory (only useful for small files) with the methods:

```
static byte[] readAllBytes(Path path)
                              throws IOException
static List<String> readAllLines(Path path)
                              throws IOException
static List<String> readAllLines(Path path, Charset cs)
                              throws IOException
```

For example:

```
try {
    // By default it uses StandardCharsets.UTF_8
    List<String> lines = Files.readAllLines(
                             Paths.get("file.txt"));
    lines.forEach(System.out::println); }
} catch (IOException e) { /** */ }
```

Or to read a file in an efficient way:

```
static BufferedReader newBufferedReader(Path path)
        throws IOException
static BufferedReader newBufferedReader(Path path, Charset cs)
        throws IOException
```

For example:

```
Path path = Paths.get("/temp/dir1/files.txt");
// By default it uses StandardCharsets.UTF_8
```

```
try (BufferedReader reader = Files.newBufferedReader(path,
        StandardCharsets.ISO_8859_1)) {
    String line = null;
    while((line = reader.readLine()) != null)
        System.out.println(line);
} catch (IOException e) { /** ... */ }
```

The `Files` class has two methods to delete files/directories.

```
static void delete(Path path) throws IOException
```

It removes the file/directory or throws an exception if something fails:

```
try {
    Files.delete(Paths.get("/temp/dir1/file.txt"));
    Files.delete(Paths.get("/temp/dir1"));
} catch (NoSuchFileException nsfe) {
    // If the file/directory doesn't exists
} catch (DirectoryNotEmptyException dnee) {
    // To delete a directory, it must be empty,
    // otherwise, this exception is thrown
} catch (IOException ioe) {
    // File permission or other problems
}
```

The second method is:

```
static boolean deleteIfExists(Path path) throws IOException
```

This method returns `true` if the file was deleted or false if the file could not be removed because it did not exist, in other words, unlike the first method, this doesn't throw a `NoSuchFileException` (but it still throws a `DirectoryNotEmptyException` and an `IOException` for other problems):

```
try {
    Files.delete(Paths.get("/temp/dir1/file.txt"));
} catch (DirectoryNotEmptyException dnee) {
    // To delete a directory, it must be empty,
} catch (IOException ioe) {
    // File permission or other problems
}
```

To copy files/directories, we have the method:

```
static Path copy(Path source, Path target,
                 CopyOption... options) throws IOException
```

It returns the path to the target file, and when copying a directory, its content won't be copied.

By default, the copy fails if the destination file already exists. Also, file attributes won't be copied, and when copying a symbolic link, its target will be copied.

We can customize this behavior with the following `CopyOption` enums:

- **StandardCopyOption.REPLACE_EXISTING**
  Performs the copy when the target already exists. If the target is a symbolic link, the link itself is copied and If the target is a non-empty directory, a `FileAlreadyExistsException` is thrown.

- **StandardCopyOption.COPY_ATTRIBUTES**
  Copies the file attributes associated with the file to the target file. The exact attributes supported are file system and platform dependent, except for last-modified-time, which is supported across platforms.
- **LinkOption.NOFOLLOW_LINKS**
  Indicates that symbolic links should not be followed, just copied.

Here's an example:

```
import static java.nio.file.StandardCopyOption. REPLACE_EXISTING;
...
try {
    Files.copy(Paths.get("in.txt"),
               Paths.get("out.txt"),
               REPLACE_EXISTING);
} catch (IOException e) { /** ... */ }
```

There are methods to copy between a stream and a `Path` also:

```
static long copy(InputStream in, Path target,
                 CopyOption... options) throws IOException
```

Copies all bytes from an input stream to a file. By default, the copy fails if the target already exists or is a symbolic link. If the `StandardCopyOption.REPLACE_EXISTING` option is specified, and the target file already exists, then it is replaced if it's not a non-empty directory. If the target file exists and is a symbolic link, then the symbolic link is replaced. Actually, in Java 8, the `REPLACE_EXISTING` option is the only option required to be supported by this method.

```
static long copy(Path source,
                 OutputStream out) throws IOException
```

Copies all bytes from a file to an output stream.

For example:

```
try (InputStream in = new FileInputStream("in.csv");
        OutputStream out = new FileOutputStream("out.csv")) {
    Path path = Paths.get("/temp/in.txt");
    // Copy stream data to a file
    Files.copy(in, path);
    // Copy the file data to a stream
    Files.copy(path, out);
} catch (IOException e) { /** ... */ }
```

To move or rename a file/directory, we have the method:

```
static Path move(Path source, Path target,
                 CopyOption... options) throws IOException
```

By default, this method will follow links, throw an exception if the file already exists, and not perform an atomic move.

We can customize this behavior with the following `CopyOption` enums:

- **StandardCopyOption.REPLACE_EXISTING**
  Performs the move when the target already exists. If the target is a symbolic link, only the link itself is moved.

- **StandardCopyOption.ATOMIC_MOVE**
  Performs the move as an atomic file operation. If the file system does not support an atomic move, an exception is thrown.

This method can move a non-empty directory. However, if the target exists, trying to move a non-empty directory will throw a `DirectoryNotEmptyException`. This exception will also be thrown when trying to move a non-empty directory across a drives or partitions.

For example:

```
try {
    // Move or rename dir1 to dir2
    Files.move(Paths.get("c:\\temp\\dir1|"),
               Paths.get("c:\\temp\\dir2");
} catch (IOException e) { /** ... */ }
```

Managing metadata

When talking about a file system, metadata give us information about a file or directory, like its size, permissions, creation date, etc. This information is referred as attributes, and some of them are system-dependent.

The `Files` class has some methods to get or set some attributes from a `Path` object:

```
static long size(Path path) throws IOException
```

Returns the size of a file (in bytes).

```
static boolean isDirectory(Path path, LinkOption... options)
```

Tests whether a file is a directory.

```
static boolean isRegularFile(Path path, LinkOption... options)
```

Tests whether a file is a regular file.

```
static boolean isSymbolicLink(Path path)
```

Tests whether a file is a symbolic link.

```
static boolean isHidden(Path path) throws IOException
```

Tells whether a file is considered hidden.

```
static FileTime getLastModifiedTime(Path path,
            LinkOption... options) throws IOException
static Path setLastModifiedTime(Path path,
            FileTime time) throws IOException
```

Returns or updates a file's last modified time.

```
static UserPrincipal getOwner(Path path,
            LinkOption... options) throws IOException
```

```
static Path setOwner(Path path,
            UserPrincipal owner) throws IOException
```

Returns or updates the owner of the file.

In methods that take an optional `LinkOption.NOFOLLOW_LINKS` , symbolic links are not followed (by default they are).

In the case of `getLastModifiedTime()` and `setLastModifiedTime()` the class `java.nio.file.attribute.FileTime` represents the value of a file's time stamp attribute.

We can create an instance of `FileTime` with these `static` methods:

```
static FileTime from(Instant instant)
static FileTime from(long value, TimeUnit unit)
static FileTime fromMillis(long value)
```

And from a `FileTime` we can get an `Instant` or milliseconds as `long` :

```
Instant toInstant()
long toMillis()
```

For example:

```
try {
    Path path = Paths.get("/temp/dir1/file.txt");
    FileTime ft = Files.getLastModifiedTime(path);
    Files.setLastModifiedTime(path,
        FileTime.fromMillis(ft.toMillis + 1000)); // adds a second
} catch (IOException e) { /** ... */ }
```

In the case of `getOwner()` and `setOwner()` the interface `java.nio.file.attribute.UserPrincipal` is an abstract representation of an identity that can be used like this:

```
try {
    Path path = Paths.get("/temp/dir1/file.txt");
    // FileSystems.getDefault() also gets a FileSystem object
    UserPrincipal owner = path.getFileSystem()
                .getUserPrincipalLookupService()
                .lookupPrincipalByName("jane");
    Files.setOwner(path, owner);
} catch (IOException e) { /** ... */ }
```

These methods are useful to get or update a single attribute. But we can also get a group of related attributes by functionality or by a particular systems implementation as a view.

The three most common view classes are:

- **java.nio.file.attribute.BasicFileAttributeView**
  Provides a view of basic attributes supported by all file systems.
- **java.nio.file.attribute. DosFileAttributeView**
  Extends `BasicFileAttributeView` to support additionally a set of DOS attribute flags that are used to indicate if the file is read-only, hidden, a system file, or archived.
- **java.nio.file.attribute. PosixFileAttributeView**
  Extends `BasicFileAttributeView` with attributes supported on POSIX systems,

such as Linux and Mac. Examples of these attributes are file owner, group owner, and related access permissions.

You can get a file attribute view of a given type to read or update a set of attributes with the method:

```
static <V extends FileAttributeView> V getFileAttributeView(
            Path path, Class<V> type,LinkOption... options)
```

For example, `BasicFileAttributeView` has only one update method:

```
try {
   Path path = Paths.get("/temp/dir/file.txt");
   BasicFileAttributeView view =
      Files.getFileAttributeView(path,
                 BasicFileAttributeView.class);
   // Get a class with read-only attributes
   BasicFileAttributes readOnlyAttrs =
                   view.readAttributes();
   FileTime lastModifiedTime =
                   FileTime.from(Instant.now());
   FileTime lastAccessTime =
                   FileTime.from(Instant.now());
   FileTime createTime =
                   FileTime.from(Instant.now());
   //If any argument is null,
   //the corresponding value is not changed
   view.setTimes(lastModifiedTime,
                 lastAccessTime,
                 createTime);
} catch (IOException e) { /** ... */ }
```

Most of the time, you'll work with the read-only versions of the file views. In this case, you can use the following method to get them directly:

```
static <A extends BasicFileAttributes> A
   readAttributes(Path path, Class<A> type,
                  LinkOption... options)
        throws IOException
```

The second parameter is the return type of the method, the class that contains the attributes to use (notice that all attributes classes extend from `BasicFileAttributes` because it contains attributes common to all file systems). The third argument is when you want to follow symbolic links.

Here's an example of how to access the file attributes of a file using the `java.nio.file.attribute.BasicFileAttributes` class:

```
try {
   Path path = Paths.get("/temp/dir1/file.txt");
   BasicFileAttributes attr = Files.readAttributes(
         path, BasicFileAttributes.class);
   // Size in bytes
   System.out.println("size(): " + attr.size());
   // Unique file identifier (or null if not available)
   System.out.println("fileKey(): " + attr.fileKey());

   System.out.println("isDirectory(): " + attr.isDirectory());
   System.out.println("isRegularFile(): " + attr.isRegularFile());
   System.out.println("isSymbolicLink(): " + attr.isSymbolicLink());
   // Is something other than a file, directory, or symbolic link?
   System.out.println("isOther(): " + attr.isOther());
```

```
    // The following methods return a FileTime instance
    System.out.println("creationTime(): " + attr.creationTime());
    System.out.println("lastModifiedTime():"+attr.lastModifiedTime());
    System.out.println("lastAccessTime(): " + attr.lastAccessTime());
} catch (IOException e) { /** ... */ }
```

Key Points

- The primary classes of `java.nio.file` are `Path`, `Paths`, and `Files`. They are intended to be a replacement of the `java.io.File` class.
- The `Path` interface defines an object that represents the path to a file or a directory.
- `java.nio.file.Paths` provides methods to create a `Path` object.
- The absolute path representation of a `Path` object has a root component (either `c:\` or `/`) and a sequence of names separated by a (forward or backward) slash.
- The `Path` interface has methods get the elements of the path, normalize paths, and get attributes of the path (isAbsolute(), getFileSystem(), etc), among others. It also implements `Comparable` and `equals()` to test for equality.
- The `java.nio.file.Files` class has static methods for common operations on files and directories. In contrast with the `java.io.File class`, all methods of `Files` work with `Path` objects.
- Examples of these operations are checking the existence of a file, copying, moving, deleting, and reading.
- You can also get attributes of a file individually (with methods like `isHidden()`) or in a group through views.
- The three most common view classes are `BasicFileAttributeView`, `DosFileAttributeView`, and `PosixFileAttributeView`.
- You can get a file attribute view of a given type to read or update a set of attributes with the method `getFileAttributeView()`.
- You can get a class that is a read-only version of the view with the method `readAttributes()`.

Self Test

1. Given:

```
Path path1 = Paths.get("/projects/work/../fun");
Path path2 = Paths.get("games");
System.out.println(path1.resolve(path2));
```

Which of the following is the result of executing the above lines?

A. `/project/work/fun/games`
B. `/project/fun/games`
C. `/project/work/../fun/games`
D. `games`

2. Given:

```
Path path = Paths.get("c:\\Users\\mark");
```

Which of the following will return `Users` ?

A. `path.getRoot()`
B. `path.getName(0)`
C. `path.getName(1)`
D. `path.subpath(0, 0);`

3. Which of the following is not a valid `CopyOption` for `Files.copy()` ?

A. `NOFOLLOW_LINKS`
B. `REPLACE_EXISTING`
C. `ATOMIC_MOVE`
D. `COPY_ATTRIBUTES`

4. Given:

```
Path path =
  Paths.get("c:\\.\\temp\\data\\..\\.\\dir\\..\\file.txt");
try {
   path = path.toRealPath();
} catch (IOException e) { }
System.out.println(path.subpath(1,2));
```

Which is the result?

A. `temp`
B. `data`
C. `dir`
D. `file.txt`

5. Which of the following is a valid way to set a file's create time?

A.

```
FileTime time = FileTime.from(Instance.now());
Files.getFileAttributeView(path,
      BasicFileAttributeView.class)
   .setTimes(null, time, null);
```

B.

```
Files.setCreateTime(path,
   FileTime.from(Instance.now());
```

C.

```
Files.getFileAttributeView(path,
      BasicFileAttributeView.class)
   .setTimes(null, null, Instance.now());
```

D.

```
FileTime time = FileTime.from(Instance.now());
Files.getFileAttributeView(path,
      BasicFileAttributeView.class)
   .setTimes(null, null, time);
```

Open answers page

23. Java I/O Fundamentals 25. Files and Streams